ABC: Algebraic Bound Computation for Loops*

Régis Blanc[1], Thomas A. Henzinger[2], Thibaud Hottelier[3], and Laura Kovács[4]

[1] EPFL
[2] IST Austria
[3] UC Berkeley
[4] TU Vienna

Abstract. We present ABC, a software tool for automatically computing symbolic upper bounds on the number of iterations of nested program loops. The system combines static analysis of programs with symbolic summation techniques to derive loop invariant relations between program variables. Iteration bounds are obtained from the inferred invariants, by replacing variables with bounds on their greatest values. We have successfully applied ABC to a large number of examples. The derived symbolic bounds express non-trivial polynomial relations over loop variables. We also report on results to automatically infer symbolic expressions over harmonic numbers as upper bounds on loop iteration counts.

1 Introduction

Establishing tight upper bounds on the execution times of programs is both difficult and interesting, see e.g. [10,5,9,8]. We present ABC, a new software tool for automatically computing tight symbolic upper bounds on the number of iterations of nested program loops. The derived bounds express polynomial relations over loop variables. ABC is fully automatic, combines static analysis of programs with symbolic summation techniques, and requires no user-guidance in providing additional set of predicates, templates and assertions. ABC is also able to derive symbolic expressions over harmonic numbers as upper bounds on loop iteration counts, which, to the best of our knowledge, is not yet possible by other works.

In our approach to bound computation, we have identified a special class of nested loop programs, called the *ABC-loops* (Section 3.1). Further, we have built a *loop converter* to transform, whenever possible, arbitrary loops into their equivalent ABC-loop format (Section 3.2). Informally, an ABC-loop is a nested for-loop such that each loop from the nested loop contains exactly one iteration variable with only one condition and one (non-initializing) update on the iteration variable. For such loops, our method derives precise bounds on the number of loop iterations.

In our work, we assume that each program statement is annotated with the time units it needs to be executed. For simplicity, we assume that an iteration of an unnested loop takes one unit time, and all other instructions of the unnested loop need zero time.

The key steps of our approach to *bound computation* are as follows (Section 3.3). (i) First, we instrument the innermost loop body of an ABC-loop with a new variable

* This work was supported in part by the Swiss NSF. The fourth author is supported by an FWF Hertha Firnberg Research grant (T425-N23).

that increases at every iteration of the program. We denote this variable by z. Upper bounds on the value of z thus express upper bounds on the number of loop iterations. (ii) Next, the value of z is computed as a polynomial function over the nested loop's iteration variables. We call the relation between z and the loop's iteration variables the *z-relation*. To this end, for each loop of the ABC-loop, recurrence equations of z and the loop iteration variables are first constructed. Closed forms of variables are then derived using our *symbolic solver* which integrates special techniques from symbolic summation (Section 3.4). The derived closed forms express valid relations between z and the loop iteration variables, and thus the z-relations are loop invariant properties. (iii) Further, by replacing loop iteration variables by bounds on their greatest values in the computed z-relation, bounds on the value of z are obtained. These bounds give us tight symbolic upper bounds on the number of iterations of the program. Our method can be generalized for the timing analysis of loops whose iteration bounds involve harmonic expressions over the loop variables (Section 3.5).

Implementation. ABC was implemented in the the Scala programming language [18], contains altogether 5437 lines of Scala code, and is available at:

`http://mtc.epfl.ch/software-tools/ABC/`

Inputs to ABC are loops written in the Scala syntax. ABC first rewrites the input loop into an equivalent ABC-loop by using its *loop converter*, and then computes bounds on loop iteration counts using its *bound computer*. The *bound computer* relies on the *symbolic solver* in order to derive closed forms of symbolic sums and simplify mathematical expressions. The overall workflow of ABC is given in Figure 1.

Note that *ABC does not rely on an external computer algebra package for symbolic summation.*

Experiments. We successfully applied ABC on examples from [10,9], as well as on 90 nested loops extracted from the JAMA package [13] – see Section 4 and the mentioned URL[1]. Altogether, we ran ABC on 558 lines of JAMA. ABC computed precise upper bounds on iteration counts for all loops, and inferred the z-relation for 87 loops, all in less than one second on a machine with a 2.8 GHz Intel Core 2 Duo processor and 2GB of RAM. The 3 loops for which ABC was not able to derive the z-relation were actually sequences of loops.

We believe that similar experimental results as the ones resulting from JAMA could be obtained by running ABC on the Jampack library [20], or on various numerical packages of computer algebra packages such as Mathematica [22], Matlab [4], or Mathcad [2].

Related work. We only discuss some of the many methods that are related to ABC.

Paper [15] infers polynomial loop invariants among program variables by using polynomial invariant templates of bounded degree. Unlike [15], where no restrictions on the considered loops were made, we require no user guidance in providing invariant templates but automatically derive invariants (z-relations) for a restricted class of loops. Our method has thus advantage in automation, but it is restricted to ABC-loops.

[1] There are 167 loops in JAMA amongst which there are 90 nested for-loops. ABC successfully inferred the exact bound for all but three for-loops.

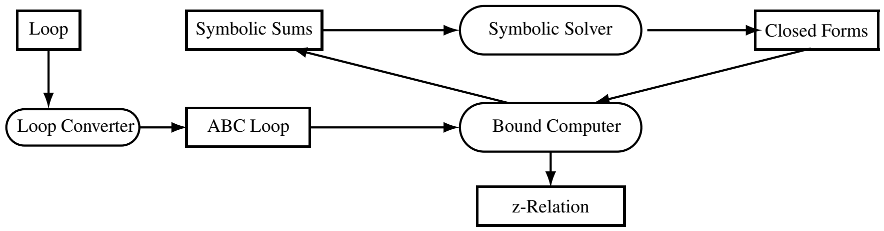

Fig. 1. The ABC tool

The approach presented in [11] infers invariants and bound assertions for loops with nested conditionals and assignments, where the assignments statements describe non-trivial recurrence relations over program variables (i.e. variable initializations are not allowed). To this end, loops are first represented by a collection of loop-free program paths, where each path corresponds to one conditional branch. Further, recurrence solving over variables is applied on each program path separately. Bounds on iteration counters can be finally inferred if the iteration counters are changed by each path in the same manner. Due to these restrictions, nested loops cannot be handled in [11]. Contrarily to [11], we infer bound assertions as z-relations for nested loops, but, unlike [11], our invariant assertions are only over loop iteration variables and not arbitrary program variables.

Paper [10] derives iteration bounds of nested loops by pattern matching simple recurrence equations. Contrarily to [10], we solve more general recurrence equations using the Gosper algorithm [6] and identities over harmonic numbers [7].

Solving recurrence relations is also the key ingredient in [1] for computing bounds. Unlike our method, evaluation trees for the unfoldings of the recurrence relations are first built in [1], and closed forms of recurrences are then derived from the maximum size of the trees. Contrarily to [1], we can handle more general recurrences by means of symbolic computation, but [1] has the advantage of solving non-deterministic recurrences that may result from the presence of guards in the loop body.

Symbolic upper bounds on iteration counts of multi-path loops are automatically derived in [8]. The approach deploys control-flow refinement methods to eliminate infeasible loop paths and rewrites multi-path loops into a collection of simpler loops for which bound assertions are inferred using abstract interpretation techniques [3]. The programs handled by [8] are more general than the ABC-loops. Unlike [8], we do not rely on abstract interpretation, and are able to infer harmonic expressions as upper bounds on loop iterations counts. Abstract interpretation is also used in [12,16] for automatically inferring upper and lower bounds on the number of execution steps of logic programs.

Paper [14] describes an automated approach for inferring linear upper bounds for functional programs, by solving constraints that are generated using linear programming. In our work we derive polynomial, and not just linear, upper bounds.

There has been a great deal of research on estimating the worst case execution time (WCET) of real-time systems, see e.g. [5,9,19]. Papers [5,9] automatically infer loop

```
for ($i = 1$; $i \leq n$; $i = i + 1$) do
 for ($j = 1$; $j \leq n$; $j = j + 1$) do
  skip
 end do
end do
```

(a)

```
for ($i = 0$; $i \leq n$; $i = i + 1$) do
 for ($j = 0$; $j \leq m$; $j = j + 2$) do
  skip
 end do
end do
```

(b)

Fig. 2. Examples illustrating the power of ABC to (i) compute z-relations as loop invariants, and (ii) infer tight upper bounds on the number of iterations of loops

bounds only for simple loops; bounds for the iteration numbers of multi-path loops must be provided as user annotations. The aiT tool [5] determines the number of loop iterations by relying on a combination of interval-based abstract interpretation with pattern matching on typical loop patterns. The SWEET tool [9] determines upper bounds on loop iterations by unrolling loops dynamically and analyzing each loop iteration separately using abstract interpretation. In contrast, our method is fully automatic and path-insensitive, but it is restricted to ABC-loops. The TuBound tool [19] implements a constraint logic based approach for loop analysis to compute tight bounds for nested-loops. The constraints being solved in [19] are directly obtained from the loop conditions and express bounds on the loop iteration variables. Unlike [19], we infer loop bounds by computing closed forms of iteration variables.

2 Motivating Examples

We first give some examples illustrating what kind of iteration bounds ABC can automatically generate.

Consider Figure 2(a) taken from the JAMA library [13]. ABC first instruments the innermost loop of Figure 2(a) with a new variable z, initialized to 1, for counting the number of iterations of Figure 2(a). The thus obtained loop is presented in Figure 3(a). Further, by applying ABC on Figure 3(a), we derive the z-relation[2]:

$$z = (i - 1)n + j$$

as an invariant property of the loop. By replacing i and j with bounds on their greatest values (i.e. n) in the z-relation, the number of iterations of Figure 2(a) is bounded by:

$$n^2.$$

Consider next Figure 2(b) with a non-unit increment, and its "instrumented" version in Figure 3(b). We obtain the z-relation:

$$z = 1 + \left\lfloor \frac{j}{2} \right\rfloor + i\left(\left\lfloor \frac{m}{2} \right\rfloor + 1\right),$$

[2] Actually, the loops of Figure 2 are first translated into their equivalent ABC-format, and then the z variable is introduced in their innermost loop body. For simplicity, in Figure 3 we present the "instrumentation" step directly on the loops of Figure 2 and not on their ABC-loop formats.

```
z = 1
for (i = 1; i ≤ n; i = i + 1) do
 for (j = 1; j ≤ n; j = j + 1) do
  z = z + 1
 end do
end do
```

(a)

```
z = 1
for (i = 0; i ≤ n; i = i + 1) do
 for (j = 0; j ≤ m; j = j + 2) do
  z = z + 1
 end do
end do
```

(b)

Fig. 3. Figure 2 instrumented by ABC

yielding:

$$1 + (1 + n)\left\lfloor \frac{m}{2} \right\rfloor + n$$

as a tight upper bound on loop iteration counts[3], where $\lfloor \frac{m}{2} \rfloor$ denotes the largest integer not greater than $\frac{m}{2}$.

In the sequel, we illustrate the main steps of ABC on Figure 2(b).

3 ABC: System Description

We have identified a special class of loops, called the *ABC-loops* (Section 3.1), and designed a loop converter for translating programs into their equivalent ABC-loop shape (Section 3.2). Algorithmic methods from symbolic summation, implemented in our symbolic solver (Section 3.4), are further deployed in ABC to automatically derive upper bounds on loop iterations of ABC-loops (Section 3.3).

3.1 ABC-Loops

We denote by $\mathbb{Z}$ the set of integer numbers, and by $\mathbb{Z}[x]$ the ring of polynomial functions in indeterminate x over $\mathbb{Z}$.

We consider programs of the following form:

```
for ($i_1 = 1;\ i_1 \le c;\ i_1 = i_1 + 1$) do
  for ($i_2 = 1;\ i_2 \le f_1(i_1);\ i_2 = i_2 + 1$) do
    ...
    for ($i_d = 1;\ i_d \le f_{d-1}(i_1, \ldots, i_{d-1});\ i_d = i_d + 1$) do
      skip                                                        (1)
    end do
    ...
  end do
end do
```

where $i_1, \ldots, i_d$ are pairwise disjoint *scalar variables* (called *loop iteration variables*) with values from $\mathbb{Z}$, c is an integer-valued symbolic constants, and $f_k \in \mathbb{Z}[i_1, \ldots, i_k]$ are polynomial functions ($k = 1, \ldots, d-1$).

[3] In our work we did not consider analyzing the relations between the smallest and greatest symbolic values of the loop iteration variables. It may however be the case that these symbolic values are such that the loops are never executed (e.g. $n < 0$).

Algorithm 1. Loop Converter

```
Input: For-loop $F$ and $conversion\_list = \{\}$
Output: ABC-loop $F'$ and $conversion\_list$
1: $\langle ovar,\ oincr\rangle :=$ ⟨outer_iteration_variable($F$), outer_iteration_increment($F$)⟩
2: $\langle olbound,\ oubound\rangle :=$ ⟨outer_iteration_lowerbound($F$), outer_iteration_upperbound($F$)⟩
3: $nvar :=$ fresh_variable()
4: $F_0 := loop\_body(F)\big[ovar \mapsto oincr \cdot \big(nvar + olbound - 1\big)\big]$
5: $conversion\_list := conversion\_list \cup \{ovar = oincr \cdot \big(nvar + olbound - 1\big)\}$
6: if isloop($F_0$) then
7:     $F' :=$ for-loop($nvar$, 1, $\lfloor \frac{oubound - olbound}{oincr} \rfloor + 1$, 1, Loop Converter($F_0$))
8: else
9:     $F' :=$ for-loop($nvar$, 1, $\lfloor \frac{oubound - olbound}{oincr} \rfloor + 1$, 1, $F_0$)
10: end if
```

In what follows, loops satisfying (1) will be called *ABC-loops*.

3.2 The Loop Converter

Converting loops into ABC-loops is done as presented in Algorithm 1. The algorithm (i) converts loops into equivalent ones such that the smallest values of the loop iteration variables are 1, and (ii) converts loops with arbitrary increments over the iteration variables into equivalent loops with increments of 1. The for-loop(v,e_1,e_2,e_3,body) notation used in Algorithm 1 is a short-hand notation for the loop:

$$\underline{\textbf{for}}\ (v = e_1; v \le e_2; v = v + e_3)\ \underline{\textbf{do}}\ \textit{body}\ \underline{\textbf{end do}}.$$

In more detail, Algorithm 1 takes as input a nested for-loop F and an empty list $conversion\_list$, and returns, whenever possible, an ABC-loop F' that is equivalent to F. The $conversion\_list$ is used to store the list of changes made by Algorithm 1 on the iteration variables of F.

Lines 4-9 of Algorithm 1 are required to convert F into an equivalent loop whose outermost loop has the following properties: it iterates over a new variable $nvar$ instead of the iteration variable $ovar$ of the outermost loop of F, where $nvar$ and $ovar$ are polynomially related; the smallest value of $nvar$ is 1 (instead of the smallest value $olbound$ of $ovar$); $nvar$ is increment by 1 (instead of the $oincr$ increment value of $ovar$); the greatest value of $nvar$ is given by the largest integer not greater than the rational expression $\frac{oubound - olbound}{oincr} + 1$, where $oubound$ is the greatest value of $ovar$. The appropriately modified[4] loop body F_0 of F is processed in the similar manner, yielding finally the ABC-loop F' that is equivalent to F.

Example 1. Consider Figure 2(b). By applying Algorithm 1, the loop iteration variables i_1 and j_1 are introduced with $i = i_1 - 1$ and $j = 2(j_1 - 1)$ (lines 3-5 of Algorithm 1). The smallest values of i_1 and j_1 are 1, their greatest values are respectively $n + 1$ and $\lfloor \frac{m}{2} \rfloor + 1$, and i_1 and j_1 are incremented by 1 (lines 6-9 of Algorithm 1). The ABC-loop format of Figure 2(b) is given in Figure 4(a).

[4] The expression $s[x \mapsto e]$ denotes the expression obtained from s by substituting each occurrence of the variable x by the expression e.

```
for ($i_1 = 1;\ i_1 \leq n+1;\ i_1 = i_1+1$) do
  for($j_1 = 1;\ j_1 \leq \lfloor \frac{m}{2} \rfloor + 1;\ j_1 = j_1+1$) do
    skip
  end do
end do
            (a)
```

```
$z = 1$
for ($i_1 = 1;\ i_1 \leq n+1;\ i_1 = i_1+1$) do
  for ($j_1 = 1;\ j_1 \leq \lfloor \frac{m}{2} \rfloor + 1;\ j_1 = j_1+1$) do
    $z := z+1$
  end do
end do
            (b)
```

Fig. 4. ABC-loop format of Figure2(b) and its instrumented version, where $i = i_1 - 1$ and $j = 2(j_1 - 1)$. Note that $\lfloor m/2 \rfloor \in \mathbb{Z}$.

Based on Algorithm 1 and keeping the notations of (1), we conclude that the general shape of loops that can be converted into ABC-loops is:

```
for ($i_1 = l;\ i_1 \leq c;\ i_1 = i_1 + inc_1$) do
  for ($i_2 = g_1(i_1);\ i_2 \leq f_1(i_1);\ i_2 = i_2 + inc_2$) do
    ...
    for ($i_d = g_{d-1}(i_1, \ldots, i_{d-1});\ i_d \leq f_{d-1}(i_1, \ldots, i_{d-1});\ i_d = i_d + inc_d$) do
      skip                                                                    (2)
    end do
    ...
  end do
end do
```

where l, $inc_1, \ldots, inc_d$ are integer-valued symbolic constants, and $g_k \in \mathbb{Z}[i_1, \ldots, i_k]$.

3.3 The Bound Computer

We assume that each program statement is annotated with the time units it needs to be executed. For simplicity, we assume that an iteration of an unnested ABC-loop takes one time unit, and all other instructions of the unnested loop need zero time (e.g. assignment statements take zero time to be executed). That is we compute a bound on the total number of loop iterations of an ABC-loop (1).

In our approach to bound computation, we instrument the innermost loop body of (1) with a new variable that increases at every iteration of the program, and is initialized to 1 before entering the ABC-loop. We denote this variable by z. From (1), we thus obtain:

```
$z = 1$
for ($i_1 = 1;\ i_1 \leq c;\ i_1 = i_1 + 1$) do
  ...
  for ($i_d = 1;\ i_d \leq f_{d-1}(i_1, \ldots, i_{d-1});\ i_d = i_d + 1$) do
    $z := z + 1$                                                              (3)
  end do
  ...
end do
```

Example 2. The instrumented loop of Figure 4(a) is given in Figure 4(b).

Upper bounds on the value of z give upper bounds on the number of iterations of (3). We are hence left with computing the value of z as a function, called the *z-relation*,

Algorithm 2. Bound Computer

```
Input: ABC-loop $F$, initial value $z_0$ of $z$
Output: $z$-relation $zrel$
 1: $inner$ := loop_body($F$)
 2: $incr$ := z_reduce_loop($inner$)
 3: $\langle ovar, oubound \rangle$ := $\langle$outer_iteration_variable($F$),outer_iteration_upperbound($F$)$\rangle$
 4: $nvar$ := fresh_variable()
 5: $z_i$ :=$z_0$ + solve_sum($nvar$, 1, $ovar - 1$, $incr[ovar \mapsto nvar]$)
 6: if isloop(inner) then
 7:     $zrel$ := $z$ =Bound Computer($inner$, $z_i$)
 8: else
 9:     $zrel$ :=$z = z_i$
10: end if
```

over $i_1, \ldots, i_d$. To this end, the value of z at an arbitrary iteration of the outermost loop of (3) is first computed.

Computing the value of z after an arbitrary iteration of the outermost loop of (3). Let us consider a more general loop than (3):

```
for ($i_1 = 1;\ i_1 \leq c;\ i_1 = i_1 + 1$) do
  for ($i_2 = 1;\ i_2 \leq f_1(i_1);\ i_2 = i_2 + 1$) do
    ...
    for ($i_d = 1;\ i_d \leq f_{d-1}(i_1, \ldots, i_{d-1});\ i_d = i_d + 1$) do
      $z := z + g(i_d)$                                              (4)
    end do
    ...
  end do
end do
```

where $i_1, \ldots, i_d$, c, $f_1, \ldots, f_{d-1}$ are as in (1), and $g \in \mathbb{Z}[i_d]$. In particular, if $g = 1$ then (4) becomes (3).

Let $s_1, \ldots, s_l$ be nonnegative integers ($l = 1, \ldots, d$) such that $1 \leq s_1 \leq c$, $1 \leq s_2 \leq f_1(i_1)$, …, and $1 \leq s_l \leq f_{l-1}(i_1, \ldots, i_{l-1})$. In the sequel we consider $s_1, \ldots, s_l$ arbitrary but fixed. We write $x^{(l,\|s_1,\ldots,s_l\|)}$ to mean the value of a variable $x \in \{i_1, \ldots, i_d, z\}$ in (4) such that the kth loop of (4) is at its s_kth iteration ($k = 1, \ldots, l$),

We are thus interested in deriving $z^{(1,\|s_1\|)}$ for $s_1 \in \{1, \ldots, c\}$. We proceed as follows. For each loop of (4), starting from the innermost one, we (i) model the assignment over z as a recurrence equation, (ii) deploy symbolic summation algorithms to compute the closed form of z, and (iii) replace the loop by a single assignment over z expressing the relation between the values of z before the first and after the last execution of the loop. Steps (i)-(iii) are recursively applied until all loops of (4) are eliminated.

In more detail, $z^{(1,\|s_1\|)}$ is derived as follows. We start with the innermost loop of (4). The assignment over z is modeled by the recurrence equation:

$$z^{(d,\|s_1,\ldots,s_d+1\|)} = z^{(d,\|s_1,\ldots,s_d\|)} + g(i_d^{(d,\|s_1,\ldots,s_d-1\|)}), \tag{5}$$

yielding:

$$z^{(d,\|s_1,\dots,s_d\|)} = ini_z + \sum_{k=1}^{s_d} g(i_d^{(d,\|s_1,\dots,k-1\|)}),$$

where $ini_z = z^{(d,\|s_1,\dots,0\|)}$ denotes the value of z before entering the innermost loop of (4). The value of $i_d^{(d,\|s_1,\dots,s_d\|)}$ is computed from the recurrence equation:

$$i_d^{(d,\|s_1,\dots,s_d+1\|)} = i_d^{(d,\|s_1,\dots,s_d\|)} + 1.$$

Namely, we have $i_d^{(d,\|s_1,\dots,s_d\|)} = ini_d + \sum_{k=1}^{s_d} 1$, where $ini_d = 1$ denotes the initial value of i_d (i.e. before the first iteration of the innermost loop of (4)). Hence,

$$i_d^{(d,\|s_1,\dots,s_d\|)} = s_d + 1. \tag{6}$$

Note that (6) holds for each iteration variable, that is:

$$i_l^{(l,\|s_1,\dots,s_l\|)} = s_l + 1$$

for every $l \in \{1,\dots,d\}$. For this reason, in what follows we write i_l instead of $i_l^{(l,\|s_1,\dots,s_l\|)}$ and use the relation $i_l = s_l + 1$ to speak about the value of i_l at iteration s_l of the lth loop. We thus obtain:

$$z^{(d,\|s_1,\dots,s_d\|)} = ini_z + \sum_{k=1}^{s_d} g(i_d^{(d,\|s_1,\dots,k-1\|)}) = ini_z + \sum_{k=1}^{i_d-1} g(k).$$

Since $g \in \mathbb{Z}[i_d]$, the closed form of $\sum_{k=1}^{i_d-1} g(k)$ always exists [6] and can be computed as a polynomial function over i_d (see Section 3.4).

Finally, we consider the last iteration $s_d = i_d - 1 = f_{d-1}(i_1,\dots,i_{d-1})$ of the innermost loop of (4), and write $incr_d = \sum_{k=1}^{f_{d-1}(i_1,\dots,i_{d-1})} g(k)$. We make use of $incr_d \in \mathbb{Z}[i_1,\dots,i_{d-1}]$ to “eliminate” the innermost loop of (4) and obtain:

```
for ($i_1 = 1;\ i_1 \le c;\ i_1 = i_1 + 1$) do
  ...
  for ($i_{d-1} = 1;\ i_{d-1} \le f_{d-1}(i_1,\dots,i_{d-2});\ i_{d-1} = i_{d-1} + 1$) do
    $z := z + incr_d$                                                    (7)
  end do
  ...
end do
```

Inner loops of (7) can be further eliminated by applying recursively the steps described above, since closed forms of polynomial expressions over $i_1,\dots,i_d$ yield polynomial expressions over $i_1,\dots,i_d$ whenever the summation variables are bounded by polynomial expressions. As a result, the total number of increments over z in the s_1th iteration of the outermost loop of (4) is derived. Let us denote this number by $incr_1$. Then:

$$z^{(1,\|s_1\|)} = z_0 + incr_1, \text{ where } z_0 = 1 \text{ is the value of } z \text{ before (4).}$$

Example 3. Consider Figure 4(b). We aim at deriving $z^{(1,\|s_1\|)}$, where $1 \leq s_1 \leq n+1$ is arbitrary but fixed such that $i_1 = s_1 + 1$.

From the innermost loop of Figure 4(b), we have $z^{(2,\|s_1,s_2+1\|)} = z^{(2,\|s_1,s_2\|)} + 1$ for an arbitrary but fixed s_2, where $1 \leq s_2 \leq \lfloor \frac{m}{2} \rfloor + 1$ and $j_1 = s_2 + 1$. Hence,

$$z^{(2,\|s_1,s_2\|)} = ini_2 + j_1 - 1,$$

where ini_2 is the initial value of z before entering the innermost loop. Further, after $s_2 = j_1 - 1 = \lfloor \frac{m}{2} \rfloor + 1$ iterations of the innermost loop, the total number of increments over z is:

$$incr_2 = \sum_{k=1}^{\lfloor \frac{m}{2} \rfloor + 1} 1 = \lfloor \frac{m}{2} \rfloor + 1.$$

The innermost loop of Figure 4(b) is next eliminated, yielding:

$$\underline{\textbf{for}}\ (i_1 = 1; i_1 \leq n+1; i_1 = i_1 + 1)\ \underline{\textbf{do}}\quad z = z + \lfloor \frac{m}{2} \rfloor + 1\ \underline{\textbf{end do}}$$

with the recurrence equation of z as:

$$z^{(1,\|s_1\|+1)} = z^{(1,\|s_1\|)} + \lfloor \frac{m}{2} \rfloor + 1.$$

Solving this recurrence and using that $z_0 = 1$ is the initial value of z before the outermost loop of Figure 4(b), we obtain:

$$z^{(1,\|s_1\|)} = 1 + \sum_{k=1}^{i_1 - 1} \left(\lfloor \frac{m}{2} \rfloor + 1\right) = 1 + (i_1 - 1)\left(\lfloor \frac{m}{2} \rfloor + 1\right).$$

Computing the z-relation among arbitrary values of $z, i_1, \ldots, i_d$. We are interested in deriving the value of $z^{(d,\|s_1,\ldots,s_d\|)}$, where $i_k = s_k + 1$ ($k = 1, \ldots, d$), from which *the z-relation can be immediately constructed as $z = z^{(d,\|s_1,\ldots,s_d\|)}$*.

The value $z^{(d,\|s_1,\ldots,s_d\|)}$ (and hence the z-relation) is inferred by Algorithm 2 as follows.

(a) The value $incr$ is computed such that $z^{(1,\|s_1\|)} = z_0 + incr$ (line 2 of Algorithm 2);
(b) The outermost loop of (4) is omitted (line 1 of Algorithm 2), yielding:

```
for ($i_2 = 1;\ i_2 \leq f_1(i_1);\ i_2 = i_2 + 1$) do
  ...
  for ($i_d = 1;\ i_d \leq f_{d-1}(i_1, \ldots, i_{d-1});\ i_d = i_d + 1$) do
    $z := z + g(i_d)$                                                    (8)
  end do
  ...
end do
```

(c) The value of z at the s_2th iteration of the outermost loop (8) is next computed, where the initial value of z before (8) is considered to be $z^{(1,\|s_1\|)}$ (line 7 of Algorithm 2). As a result, $z^{(2,\|s_1,s_2\|)}$ in the loop (4) is obtained.

(d) Steps (b)-(c) are recursively applied on (8) until no more loops are left and $z^{(d,\|s_1,\ldots,s_d\|)}$ is derived (lines 6-9 of Algorithm 2).

Example 4. Consider Figure 4(b). The outermost loop of Figure 4(b) is omitted (line 1 of Algorithm 2), yielding:

$$\underline{\textbf{for}}\ (j_1 = 1; j_1 \leq \lfloor \frac{m}{2} \rfloor + 1; j_1 = j_1 + 1)\ \underline{\textbf{do}}\ \ z = z + 1\ \underline{\textbf{end do}} \tag{9}$$

The total number of increments $incr_2 = \lfloor \frac{m}{2} \rfloor + 1$ over z made by (9) is computed, as presented in Example 3 (line 2 of Algorithm 2). The value $z_i = z^{(1,\|s_1\|)}$ of z at an iteration $s_1 = i_1 - 1$ of the outermost loop of Figure 4(b) is further obtained (lines 3-5 of Algorithm 2), as:

$$z_i = z_0 + \sum_{nvar=1}^{i_1-1} \big(\lfloor \frac{m}{2} \rfloor + 1\big) = 1 + (i_1 - 1)\big(\lfloor \frac{m}{2} \rfloor + 1\big).$$

Next, Algorithm 2 is called on (9) with the initial value z_i to compute the value of z at an iteration $s_2 = j_1 - 1$ of (9) (line 7 of Algorithm 2). As (9) has no inner loops, we have $incr = 1$ and $z_i' = z_i + \sum_{nvar=1}^{j_1-1} 1$, yielding (lines 2-5 of Algorithm 2):

$$z^{(2,\|s_1,s_2\|)} = z_i' = (i_1 - 1)\big(\lfloor \frac{m}{2} \rfloor + 1\big) + j_1.$$

The z-relation of Figure 4(b) is finally derived (line 9 of Algorithm 2), as:

$$z = (i_1 - 1)\big(\lfloor \frac{m}{2} \rfloor + 1\big) + j_1.$$

To obtain the z-relation of Figure 2(b), we make use of $i = i_1 - 1$ and $j = 2(j_1 - 1)$ and have:

$$z = i\left(\left\lfloor \frac{m}{2} \right\rfloor + 1\right) + \left\lfloor \frac{j}{2} \right\rfloor + 1.$$

Replacing i and j respectively with n and m in the z-relation, the upper bound on loop iteration counts of Figure 2(b) is:

$$(n + 1)\big(\left\lfloor \frac{m}{2} \right\rfloor + 1\big).$$

3.4 Symbolic Solver

Simplifying arithmetical expressions and computing closed forms of symbolic sums is performed by the symbolic solver engine of ABC. Our symbolic solver supports the closed form computation of the following sums:

$$\sum_{x=e_1}^{e_2} c_1 \cdot n_1^x \cdot x^{d_1} + \cdots + c_r \cdot n_r^x \cdot x^{d_r}$$

where e_1, e_2 are integer-valued symbolic constants, n_i, d_i are natural numbers, and c_i are rational numbers. Closed forms of such sums always exists [6]. For computing the closed forms of these sums we rely on a simplified version of the Gosper algorithm [6].

```
for $(i = 1; i \leq n; i = i + 1)$
 for $(j = 0; j \leq n; j = j + i)$
  skip
 end do
end do
```

(a) Not an ABC-loop

```
for $(i_1 = 1; i_1 \leq n; i_1 = i_1 + 1)$
 for $(j_1 = 1; j_1 \leq n_1; j_1 = j_1 + 1)$
  skip
 end do
end do
```

(b) Converted loop by ABC with $n_1 = \lfloor \frac{n}{i_1} \rfloor + 1$, and $i = i_1, j = i \cdot j_1$

```
$z := 1$
for $(i_1 = 1; i_1 \leq n; i_1 = i_1 + 1)$
 for $(j_1 = 1; j_1 \leq n_1; j_1 = j_1 + 1)$
  $z := z + 1$
 end do
end do
```

(c) Instrumented loop by ABC

Fig. 5. ABC on a non-ABC-loop

We have also instrumented our symbolic solver to handle symbolic sums whose closed forms involve harmonic numbers [7], as discussed in Section 3.5.

3.5 Beyond ABC-Loops

Our approach to bound computation implemented in ABC is complete for ABC-loops and for loops satisfying (2). That is, it *always* returns the z-relation and loop iteration bound of an ABC-loop.

It is worth to be mentioned that some loops violating (2) can still be successfully analyzed by ABC.

Consider Figure 5(a) violating (2), as updates over j depend on i. However, using Algorithm 1 we derive the loop given in Figure 5(b), yielding finally the "instrumented" loop from Figure 5(c). Further, by applying Algorithm 2, we are left with finding the closed form of $\sum_{k=1}^{i_1-1} \left\lfloor \frac{n}{i_1} \right\rfloor$. This sum cannot be further simplified [7]. However, we can compute an upper-bound on its closed form using the relation:

$$\sum_{k=1}^{i_1-1} \left\lfloor \frac{n}{i_1} \right\rfloor \leq \left\lfloor \sum_{k=1}^{i_1-1} \frac{n}{i_1} \right\rfloor = \left\lfloor n \sum_{k=1}^{i_1-1} \frac{1}{i_1} \right\rfloor .$$

Note that $\sum_{k=1}^{i_1-1} \frac{1}{i_1}$ is the harmonic number $H(i_1 - 1)$ arising from the truncation of the harmonic series [7]. We make use of $H(i_1 - 1)$ and derive an upper bound on the loop iteration count of Figure 5(a) as being a harmonic expression. To this end, we have extended our symbolic solver with some simple identities over harmonic numbers. To the best of our knowledge, inferring a tight loop bound for Figure 5(a) is not yet possible by other works.

ABC can thus be successfully applied to loops for which symbolic computation methods can be deployed to compute or approximate closed forms of loop variables. Such cases may arise from nested loops whose inner loop counters are updated by non-constant polynomial functions in the outer loop counters (i.e. yielding iteration bounds as harmonic numbers).

4 Experiments

We applied ABC to a large number of examples, including benchmark programs from recent work on timing analysis [21] as well as from the JAMA package [13].

Table 1. Experimental results obtained by ABC on benchmark examples

Loop	z-relation	Iteration bound	Time [s]
for $(i = a; i \le b; i = i + 1)$ **skip** **end do**	$z = 1 + i - a$	$1 + b - a$	0.172
for $(i = 0; i \le n - 1; i = i + 1)$ **for** $(j = 0; j \le i; j = j + 1)$ **skip** **end do** **end do**	$z = 1 + j + \frac{i+i^2}{2}$	$\frac{n+n^2}{2}$	0.219
for $(i = 1; i \le m; i = i + 1)$ **for** $(j = 1; j \le i; j = j + 1)$ **for** $(k = i + 1; k \le m; k = k + 1)$ **for** $(l = 1; l \le k; l = l + 1)$ **skip** **end do** **end do** **end do** **end do**	$z = \frac{i^2m^2-im^2+i^2m-im}{4} + \frac{i^2-i^4}{8} + \frac{i^3-i}{12} + \frac{jm+jm^2+k^2}{2} - \frac{m^2+ji^2+ji+m+k}{2} + 1$	$\frac{3m^4+2m^3-3m^2-2m}{24}$	1.281
for $(i = 0; i \le (\frac{n*n*n}{2} - 1); i = i + 1)$ **for** $(j = 0; j \le n - 1; j = j + 1)$ **for** $(k = 0; k \le j - 1; k = k + 1)$ **skip** **end do** **end do** **end do**	$z = 1 + k + \frac{in^2-in+j^2-j}{2}$	$\frac{n^5-n^4}{4}$	0.234
for $(i = 1; i \le n; i = i + 1)$ **for** $(j = 1; j \le i; j = j + 1)$ **skip** **end do** **end do**	$z = \frac{i^2-i}{2} + j$	$\frac{n^2+n}{2}$	0.203
for $(i = 1; i \le n; i = i + 1)$ **for** $(j = i; j \le n; j = j + 1)$ **skip** **end do** **end do**	$z = (i-1)n + j + \frac{i-i^2}{2}$	$\frac{n^2+n}{2}$	0.203
for $(j = 1; i \le m; j = j + 1)$ **for** $(i = 1; i \le n; i = j + 1)$ **skip** **end do** **end do**	$z = i + (j-1)n$	nm	0.187
for $(i = n; i \ge 1; i = i - 1)$ **for** $(j = m; j \ge 1; j = j - 1)$ **skip** **end do** **end do**	$z = (n-i+1)m-j+1$	nm	0.188

Tables 1 and 2 summarize some of our results obtained on a machine with a 2.8 GHz Intel Core 2 Duo processor and 2GB of RAM.

The first four programs of Table 1 are examples taken from [21], whereas the last four programs of Table 1 are loops taken from the JAMA package [13]. The examples of Table 2 are our own benchmark examples, and illustrate the power of ABC in handling nested loops whose inner loop counters polynomially depend on its outer loop counters. The difference between the first four programs of Table 2 is given by the mixed incremental/decremental updates and smallest/greatest values of the loop counters. Note that the last three programs of Table 2 yield polynomial loop bounds as sums of multivariate monomials.

Table 2. Further experimental results obtained by ABC

Loop	z-relation	Iteration bound	Time [s]

for $(i = 1; i \leq n; i = i + 1)$
for $(j = 1; j \leq m; j = j + 1)$
skip
end do
end do
	$z = j + (i - 1)m$	nm	0.187

for $(i = n; i \geq 1; i = i - 1)$
for $(j = 1; j \leq m; j = j + 1)$
skip
end do
end do
	$z = j + (n - i)m$	nm	0.187

for $(i = n; i \geq 1; i = i - 1)$
for $(j = m; j \geq 1; j = j - 1)$
skip
end do
end do
	$z = 1 - j + (n - i + 1)m$	nm	0.187

for $(i = n; i \geq 1; i = i - 1)$
for $(j = m; j \geq m; j = j - 1)$
skip
end do
end do
	$z = 1 - i - j + m + n$	n	0.203

for $(i = a; i \leq b; i = i + 1)$
for $(j = c; j \leq d; j = j + 1)$
for $(k = i - j; k \leq i + j; k = k + 1)$
skip
end do
end do
end do
	$z =$ $1 - 2ad + 2id -$ $ad^2 + id^2 + ac^2 -$ $ic^2 + j^2 - c^2 +$ $j - a + k$	$(c^2 - (d + 1)^2)(a - b - 1)$	0.328

for $(i = n; i \geq 1; i = i - 1)$
for $(j = 1; j \leq m; j = j + 1)$
for $(k = i; k \leq i + j; k = k + 1)$
for $(l = 1; l \leq k; l = l + 1)$
skip
end do
end do
end do
end do
	$z =$ $-\frac{m^2+3m+2}{4} i^2 +$ $(\frac{j^2+j-1}{2} - \frac{2m^3+9m^2+13}{12})i +$ $\frac{k^2-k}{2} + \frac{j^3-j}{6} + 1 +$ $\frac{2m^2+3mn+9m+9n+13}{12} mn$	$\frac{2m^2+3mn+9m+9n+13}{12} mn$	0.625

for $(i = n; i \geq 1; i = i - 1)$
for $(j = 1; j \leq m; j = j + 1)$
for $(k = i; k \leq p; k = k + 1)$
for $(l = q; l \leq j; l = l - 1)$
skip
end do
end do
end do
end do
	$z =$ $\frac{3j-ij-j^2+ij^2}{2} + k - l - p +$ $\frac{i^2m+im^2-i^2m^2-im}{4} +$ $jq - jk + kq - pq +$ $\frac{mn-m^2n-mn^2+m^2n^2}{4} +$ $\frac{3jp-j^2p-imp+im^2p}{2} -$ $ijq + jpq +$ $\frac{mnp-m^2np-imq}{2} - impq +$ $\frac{im^2q+mnq-mn^2q}{2} + mnpq$	$\frac{m^2n^2-mn^2-m^2n+mn}{4} +$ $\frac{mnp-m^2np+mnq-mn^2q}{2} +$ $mnpq$	0.375

The first column of Table 1 (respectively of Table 2) presents the loop being fed into ABC, the second column shows the z-relation derived by ABC, whereas the third one presents the number of loop iterations computed by ABC. The forth column gives the time (in seconds) needed by ABC to infer bounds on loop iteration counts [5]. Note that iteration bounds are *integer-valued* polynomial expressions (e.g. $n^2 + n$ is divisible by 2).

[5] Note, that the timings given in Tables 1 and 2 include also the required time to launch the JVM.

5 Conclusions

We describe the software tool ABC for automatically deriving tight symbolic upper bounds on loop iteration counts of a special class of loops, called the ABC-loops. The system was successfully tried on a large number of examples. The derived symbolic bounds express non-trivial (polynomial and harmonic) relations over loop variables.

Future work includes extending ABC to handle more complex sums, such as e.g. fractions of polynomials [17], including more sophisticated control-flow refinement techniques, such as [8], into ABC, and improving the loop converter of ABC for handling more complex loops.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 221–237. Springer, Heidelberg (2008)
2. Birkeland, B.: Calculus and Algebra with MathCad 2000. Haeftad. Studentlitteratur (2000)
3. Cousot, P., Cousot, R.: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. of POPL, pp. 238–252 (1977)
4. Danaila, I., Joly, P., Kaber, S.M., Postel, M.: An Introduction to Scientific Computing: Twelve Computational Projects Solved with MATLAB. Springer, Heidelberg (2007)
5. Ferdinand, C., Heckmann, R.: aiT: Worst Case Execution Time Prediction by Static Program Analysis. In: Proc. of IFIP Congress Topical Sessions, pp. 377–384 (2004)
6. Gosper, R.W.: Decision Procedures for Indefinite Hypergeometric Summation. PNAS 75, 40–42 (1978)
7. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete Mathematics, 2nd edn. Addison-Wesley Publishing Company, Reading (1989)
8. Gulwani, S., Jain, S., Koskinen, E.: Control-flow Refinement and Progress Invariants for Bound Analysis. In: Proc. of PLDI, pp. 375–385 (2009)
9. Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In: Proc. of RTSS, pp. 57–66 (2006)
10. Healy, C.A., Sjödin, M., Rustagi, V., Whalley, D.B., van Engelen, R.: Supporting Timing Analysis by Automatic Bounding of Loop Iterations. Real-Time Systems 18(2/3), 129–156 (2000)
11. Henzinger, T.A., Hottelier, T., Kovacs, L.: Valigator: A Verification Tool with Bound and Invariant Generation. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 333–342. Springer, Heidelberg (2008)
12. Hermenegildo, M.V., Puebla, G., Bueno, F., Lopez-Garcia, P.: Integrated Program Debugging, Verification, and Optimization using Abstract Interpretation (and the Ciao System Preprocessor). Sci. Comput. Program. 58(1-2), 115–140 (2005)
13. Hicklin, J., Moler, C., Webb, P., Boisvert, R.F., Miller, B., Pozo, R., Remington, K.: JAMA: A Java Matrix Package (2005), `http://math.nist.gov/javanumerics/jama/`
14. Jost, S., Loidl, H., Hammond, K., Scaife, N., Hofmann, M.: "Carbon Credits" for Resource-Bounded Computations Using Amortised Analysis. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 354–369. Springer, Heidelberg (2009)
15. Müller-Olm, M., Petter, M., Seidl, H.: Interprocedurally Analyzing Polynomial Identities. In: Durand, B., Thomas, W. (eds.) STACS 2006. LNCS, vol. 3884, pp. 50–67. Springer, Heidelberg (2006)

16. Navas, J., Mera, E., Lopez-Garcia, P., Hermenegildo, M.V.: User-Definable Resource Bounds Analysis for Logic Programs. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 348–363. Springer, Heidelberg (2007)
17. Nemes, I., Petkovsek, M.: RComp: A Mathematica Package for Computing with Recursive Sequences. Journal of Symbolic Computation 20(5-6), 745–753 (1995)
18. Odersky, M.: The Scala Language Specification (2008), `http://www.scala-lang.org`
19. Prantl, A., Knoop, J., Schordan, M., Triska, M.: Constraint Solving for High-Level WCET Analysis. CoRR, abs/0903.2251 (2009)
20. Stewart, G.W.: JAMPACK: A Java Package For Matrix Computations, `http://www.mathematik.hu-berlin.de/~lamour/software/JAVA/Jampack/`
21. van Engelen, R.A., Birch, J., Gallivan, K.A.: Array Data Dependence Testing with the Chains of Recurrences Algebra. In: Proc. of IWIA, pp. 70–81 (2004)
22. Wolfram, S.: The Mathematica Book. Version 5.0. Wolfram Media, Champaign (2003)